gr-pocketsdr: Bringing PocketSDR Front-Ends to GNU Radio and GNSS-SDR Ecosystem

Minhaj Uddin Ahmad MAHMAD12@CRIMSON.UA.EDU
The University of Alabama, 28 Kirkbride Lane, Tuscaloosa, AL 35487-0288

Muhammad Sami Irfan MIRFAN@CRIMSON.UA.EDU
The University of Alabama, 28 Kirkbride Lane, Tuscaloosa, AL 35487-0288

Sagar Dasgupta SDASGUPTA@UA.EDU
The University of Alabama, 28 Kirkbride Lane, Tuscaloosa, AL 35487-0288

Mizanur Rahman MIZAN.RAHMAN@UA.EDU
The University of Alabama, 28 Kirkbride Lane, Tuscaloosa, AL 35487-0288

Abstract

Open-source software-defined radio (SDR) ecosystems thrive on broad hardware support. Global Navigation Satellite System (GNSS)-SDR, a popular GNSS software receiver based on GNU Radio (GR), is no different. PocketSDR front-ends are an effective alternative to generic commodity SDRs for GNSS research. PocketSDRs are specifically tuned to GNSS bands and share a common clock across multiple radio frequency (RF) channels, enabling coherent multi-band processing. In addition, they are highly cost-effective. This paper presents gr-pocketsdr, a GR out-of-tree (OOT) module that enables open-source PocketSDR front-ends to serve as a GR signal source. Furthermore, we develop a GNSS-SDR adapter that enables use of the gr-pocketsdr module as a GNSS signal source. Experimental results show sustained IQ streaming at 20 Msps without sample loss and provide a real-time PVT (position, velocity, and time) solution. Experimental evaluation demonstrates sustained streaming at 20 Msps without sample loss. During a 328.6 s end-to-end run, GNSS-SDR produced 287 position fixes with no ring-buffer overruns. A controlled comparison with a ZeroMQ-based transport measures the processing cost, pipeline behavior, and end-to-end latency associated with crossing a process boundary. The multi-channel implementation additionally streams two independently tuned RF channels from one device while preserving their sample alignment. Finally, we demonstrate an application of the integrated receiver for characterizing the PocketSDR front-end reference oscillator. The software is released as open source to facilitate reproducible GNSS SDR research.

Proceedings of the 16th GNU Radio Conference, Copyright 2026 by the author(s).

1. Introduction

Open-source software-defined radio (SDR) frameworks provide researchers with a flexible way to combine radio-frequency hardware with programmable signal-processing chains. This flexibility is particularly useful for Global Navigation Satellite System (GNSS) research, where experiments may require access to raw intermediate-frequency or in-phase/quadrature (I/Q) samples rather than only a receiver's final positioning, velocity, and time (PVT) solution. GNU Radio provides a general framework for streaming and processing sampled radio signals, while GNSS-SDR provides an open-source software receiver built on this ecosystem (Fernandez-Prades et al., 2011).

PocketSDR is an open-source family of radio frequency (RF) front-ends specifically designed for GNSS signal processing (Takasu). The hardware combines MAX2771 GNSS receiver devices with a Cypress USB interface and is available with multiple RF channels. Unlike a general-purpose SDR, the front-end is designed around GNSS frequency bands and provides the receiver configuration needed for GNSS signal acquisition. Multiple RF channels can also share a reference oscillator, making the hardware useful for experiments involving multiple GNSS frequencies.

The usefulness of a front-end in an open-source SDR

ecosystem, however, depends not only on the hardware itself but also on how its samples enter the software processing chain. PocketSDR samples can be captured and subsequently processed by GNSS software receivers. In a capture-and-replay workflow, the user must first create a recording and then provide that recording to the receiver. This is appropriate for offline analysis and creating repeatable datasets, but it does not provide a direct path for applications that need receiver output while the RF signal is being acquired.

A native streaming interface also simplifies the relationship between front-end configuration and receiver configuration. A file-based workflow requires the sample format, sampling rate, intermediate frequency, and RF configuration to remain consistent across the acquisition and processing stages. With a native source, the front-end can be configured when the receiver starts, and the source can report the operating parameters obtained from the device itself. Sample loss can likewise be observed at the point where the USB stream enters the processing pipeline rather than inferred later from receiver behavior.

The multi-channel case introduces another consideration. Multiple PocketSDR RF channel data are contained in the same USB stream and share a reference oscillator. Separating the channels into independent processes and queues can preserve their individual sample streams but may lose coherence between them due to delays in the host operating system. A source integrated into the receiver can instead demultiplex the channels from the same USB stream and connect them directly to the corresponding signal-processing chains.

This paper presents `gr-pocketsdr`, a GNU Radio OOT module that provides a direct streaming interface to PocketSDR front-ends. The implementation receives the USB bulk stream, reconstructs the quantized samples, exposes selected RF channels as GNU Radio streams, and configures the MAX2771 devices at startup. A corresponding `Pocket_SDR_Signal_Source` integrates the source with GNSS-SDR.

The work makes four primary contributions. First, it provides a live GNU Radio source for PocketSDR hardware. Second, it integrates the source with GNSS-SDR and extends the adapter to multiple RF channels. Third, it experimentally evaluates streaming integrity and compares direct in-process operation with a ZeroMQ transport using CPU usage, pipeline lag, and end-to-end latency. Fourth, it demonstrates that the resulting receiver can be used to characterize the PocketSDR front-end's reference oscillator.

The remainder of the paper describes the front-end and integration requirements, the implementation of `gr-pocketsdr`, the experimental methodology, and the resulting measurements.

2. PocketSDR and Integration Requirements

2.1. PocketSDR Front-End

The PocketSDR front-end used in this work combines MAX2771 GNSS receiver devices with a Cypress USB interface. The MAX2771 provides the RF reception and digitization, while the USB device transfers the quantized samples to the host computer.

The front-end supports multiple RF channels. Each channel can be configured independently, while the channels can share the same reference oscillator. The resulting architecture is useful for experiments in which signals from different GNSS bands need to be processed simultaneously.

The host-side data representation differs from the complex-sample representation typically consumed by GNU Radio signal-processing blocks. The integration therefore requires a USB transport layer, sample unpacking, and channel demultiplexing before the data can be presented as GNU Radio streams.

2.2. Integration Requirements

The integration was designed around four requirements.

First, the source must support continuous streaming directly from the front-end so that GNSS-SDR can process samples while they are being acquired.

Second, the USB data must be unpacked consistently with the PocketSDR data representation. The resulting samples must preserve the sample ordering and channel assignment of the original stream.

Third, multiple RF channels must be exposed independently while retaining the alignment inherent in their common USB stream.

Fourth, the configuration applied to the hardware must be observable. A configuration file alone describes the intended state, but the source should be able to read the device registers and derive the operating parameters from the hardware itself.

These requirements define the boundary between the PocketSDR device and the GNU Radio processing graph. The resulting data path is shown in Figure 1.

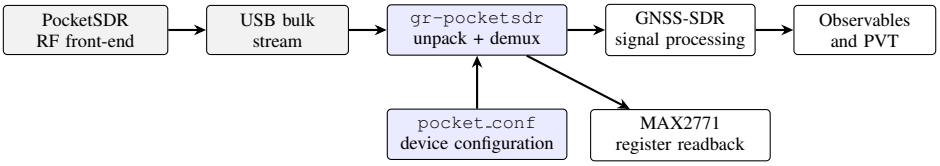

Figure 1. Live data path from the PocketSDR front-end to GNSS-SDR. The source receives the USB stream, reconstructs and demultiplexes the samples, and provides the resulting RF channels directly to the GNSS processing chain.

3. Method

3.1. USB Streaming and Sample Reconstruction

`gr-pocketsdr`[1] drives the front-end through `libusb` using a ring of six 1 MiB bulk-transfer buffers. Incoming 2- or 3-bit quantized samples are unpacked and converted to the representation expected by GNU Radio.

For the configurations evaluated in this work, the resulting sample values are ± 1 and ± 3. Channels sampled without a quadrature component are represented as $I + 0j$. One output stream is generated for each selected RF channel.

Because the channel streams are separated from the same USB byte stream, their sample ordering is preserved by construction. The source therefore does not need a separate synchronization mechanism to associate samples from different RF channels.

3.2. Device Configuration and Register Readback

The source accepts a PocketSDR configuration in `pocket_conf` format and writes the corresponding MAX2771 configuration to the device during initialization. The source subsequently reads the MAX2771 registers using `SDR_VR_REG_READ` vendor requests.

The local oscillator frequency, sampling rate, and quantization are derived from the register values returned by the device rather than being inferred solely from the configuration file. This provides a direct check of the operating state of the hardware.

To verify that the configuration reaches the device, the front-end was first configured using one `pocket_conf` file. GNSS-SDR was then started with a different configuration file without otherwise changing the hardware. The MAX2771 registers were read before and after the change. This procedure distinguishes a configuration that is merely reported by software from one that is actually applied to the RF front-end.

[1] https://github.com/minhaj6/gr-pocketsdr

3.3. GNSS-SDR Integration

The `Pocket_SDR_Signal_Source` component wraps the GNU Radio source for GNSS-SDR. The initial adapter supported a single RF channel.

GNSS-SDR's flow graph can accommodate a multi-output signal source when the source exposes multiple output streams. The original PocketSDR adapter, however, requested a single channel and connected only its first output. We extended the adapter so that the number of requested RF channels is represented explicitly and each channel can be connected to its corresponding signal conditioner.

The resulting configuration is:

```
SignalSource.implementation=Pocket_SDR_Signal_Source
SignalSource.conf_file=pocket_L1L2_8MHz.conf
SignalSource.sampling_frequency=8000000
SignalSource.RF_channels=2
SignalSource.channel0=1
SignalSource.freq0=1573420000
SignalSource.channel1=2
SignalSource.freq1=1225600000
```

Each requested channel has its own channel index and center-frequency configuration. The adapter provides the corresponding output stream to the GNSS-SDR flow graph.

3.4. Real-Sampled Configurations

The PocketSDR L1+L2 configuration used in the experiments employs real sampling with the desired signal located at an intermediate frequency of 2 MHz. In `pocket_L1L2_8MHz.conf`, CH1 uses an LO of 1573.420 MHz and CH2 uses an LO of 1225.600 MHz, both 2 MHz below their respective carrier frequencies. The sampling rate is 8 Msps and the analog bandwidth is 2.5 MHz. This configuration follows the real-sampling approach reported for PocketSDR L1 captures (Friedt).

Real sampling introduces an image that is not present in a zero-IF I/Q configuration. After the frequency-translating filter shifts the desired signal toward baseband, the corresponding mirror image occurs at -4 MHz. When the stream is decimated from 8 Msps to 4 Msps, this image folds onto DC.

The anti-alias filter must therefore reject the image before

decimation. In this configuration, the filter must stop passing frequencies by 2.75 MHz, corresponding to approximately 0.69 of the Nyquist frequency at the decimated sampling rate. The filter was designed from this folding constraint rather than from the desired passband alone.

3.5. Direct and ZeroMQ Signal Sources

Two live signal-source architectures are compared in Figure 2. The first uses `Pocket_SDR_Signal_Source`, which links `gr-pocketsdr` directly into GNSS-SDR. The second uses a GNU Radio flow graph to publish complex samples through ZeroMQ, with GNSS-SDR receiving them through `ZMQ_Signal_Source`.

Both configurations use GPS L1 C/A at 8 Msps from the same PocketSDR device configuration. The downstream GNSS-SDR configuration, beginning at the signal conditioner, is byte-identical between the two cases. The two runs are performed consecutively under the same antenna and sky conditions.

The direct source reports ring-buffer overruns when samples are discarded between the front-end and the flow graph. The ZeroMQ implementation uses a PUB/SUB transport. With `drop_on_hwm` enabled, samples can be discarded when the queue reaches its limit. In the configuration evaluated here, the GNSS-SDR adapter uses an unbounded high-water mark, so a receiving stall can instead appear as increasing queue occupancy and processing delay.

3.6. CPU and Latency Measurement

CPU and latency measurements are collected externally to GNSS-SDR. The receiver itself is not modified and no measurement block is inserted into the GNSS-SDR flow graph.

For CPU measurement, Linux process accounting provides the processor time consumed by each process through `/proc/<pid>/stat`. The `utime` and `stime` counters are sampled every two seconds. The difference between successive measurements is divided by the elapsed wall-clock time to obtain the fraction of one CPU core consumed during the interval. Results are reported as percentages, where 100 % corresponds to one fully utilized CPU core.

The CPU measurement includes all processes belonging to the corresponding processing pipeline. This is necessary for the ZeroMQ configuration because part of the acquisition and transport processing occurs in a separate process.

Two latency quantities are measured. The first is end-to-end latency, defined as the difference between the host time at which a GNSS-SDR position solution becomes available and the GNSS-derived time associated with the measurement epoch used for that solution. GNSS-SDR prints the measurement epoch in each position solution, allowing the quantity to be obtained without modifying the receiver.

The second quantity is pipeline lag. GNSS-SDR periodically reports how much signal time has been processed. If a line reporting N seconds of receiver time appears at host time $t_{\mathrm{host}}(N)$, the relative lag is

$$\mathrm{lag}(N) = (t_{\mathrm{host}}(N) - T_{\mathrm{anchor}}) - N, \tag{1}$$

where T_{anchor} is the host time associated with the beginning of the measurement. The absolute value depends on the initial amount of buffered data, so the analysis focuses on lag drift and spread.

Because the latency measurements depend on the time at which GNSS-SDR output becomes visible to the measurement process, receiver output is collected through a pseudo-terminal. Without this step, standard output buffering can delay groups of lines and produce an apparent timing drift unrelated to the signal-processing pipeline.

3.7. Reference Oscillator Characterization

The PocketSDR TCXO provides the reference for the MAX2771 and ADC. GNSS-SDR reports a receiver clock offset estimated by RTKLIB at each solution epoch. This offset is available as `user_clk_offset` in the PVT monitor message and as the fourth field of each 187-byte record in the binary PVT dump.

The clock-offset sequence is treated as a phase-error sequence relative to GNSS system time. From phase samples x_i separated by τ_0, the overlapping Allan deviation is calculated as

$$\sigma_y^2(\tau) = \frac{1}{2\tau^2(N-2m)} \sum_{i=1}^{N-2m} \left(x_{i+2m} - 2x_{i+m} + x_i\right)^2, \tag{2}$$

where $\tau = m\tau_0$.

The receiver-derived clock measurement contains both oscillator behavior and measurement noise. Assuming a typical range error of 5 m, the corresponding clock uncertainty is approximately 16.7 ns. For white phase noise, the resulting fractional-frequency noise decreases as the averaging interval increases.

The resulting receiver noise floor is summarized in Table 1. At short averaging times the receiver noise can exceed the expected instability of the front-end oscillator. The analysis therefore compares the measured Allan deviation with the estimated receiver noise floor rather than interpreting every short-term value as oscillator behavior.

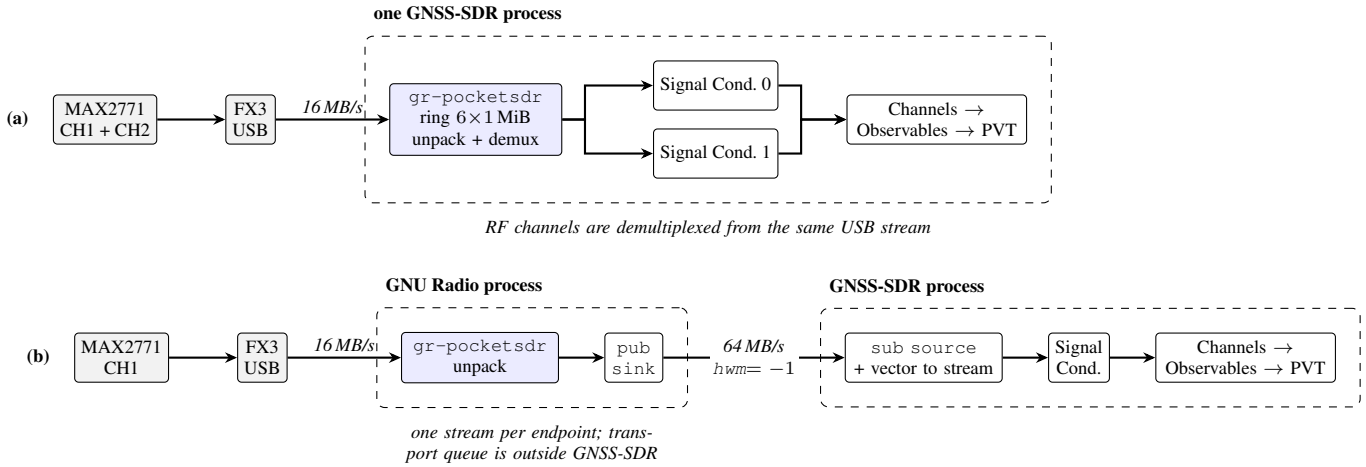

Figure 2. The two live signal-source architectures evaluated in this work. (a) The direct source connects `gr-pocketsdr` to GNSS-SDR within one process. Multiple RF channels are demultiplexed from the same USB stream. (b) The ZeroMQ architecture separates GNU Radio and GNSS-SDR into two processes and transports unpacked complex samples between them.

Table 1. Estimated receiver noise floor for a GNSS-derived clock measurement with UERE of 5 m, compared with typical TCXO stability.

τ	Receiver floor	Typical TCXO
1 s	2.9×10^{-8}	$\sim 10^{-9}$
10 s	2.9×10^{-9}	$\sim 10^{-9}$
100 s	2.9×10^{-10}	$\sim 10^{-9} \ldots 10^{-10}$
1000 s	2.9×10^{-11}	$\sim 10^{-10}$

The analysis does not bridge gaps in the clock-offset sequence as though they were valid samples. Instead, it identifies the longest uninterrupted portion of the record and reports the number of missing epochs. This prevents a data gap from being interpreted as an artificially stable oscillator.

The analysis implementation was validated using synthetic phase data containing a known 2.5 ppm frequency offset and 17 ns white phase noise. The recovered frequency offset was 2.5001 ppm. The analysis also selected the correct contiguous data span when an artificial gap was introduced.

4. Results

4.1. Configuration Verification

The register readback experiment confirmed that the configuration supplied to the source reaches the MAX2771 hardware. For CH1, the PLL divider register at address `0x04` changed from `0x0CD22C80` to `0x00082008`. The corresponding reported LO frequency changed from 1575.420 MHz to 1573.420006 MHz, while the sample rate changed from 20 Msps to 8 Msps.

The register values therefore changed in response to the configuration applied during initialization, and the subsequent source readback reflected the operating state of the hardware rather than only the contents of the configuration file.

4.2. Sustained Streaming and Real-Time PVT

The first end-to-end experiment evaluated whether the front-end could stream continuously into GNSS-SDR while the receiver produced navigation solutions.

During a 328.6 s run, CH1 was configured for GPS L1 C/A at 20 Msps using I/Q samples with 2-bit quantization. `gr-pocketsdr` reported zero ring-buffer overruns throughout the run. GNSS-SDR produced 287 position fixes, with the first fix occurring 42 s after startup and using up to seven satellites.

The absence of ring-buffer overruns indicates that the USB stream was consumed continuously during the experiment. At 20 Msps, the corresponding raw USB payload was 40 MB/s. The position solutions were generated during the acquisition run rather than from a completed recording, demonstrating the intended live processing path.

4.3. Anti-Alias Filtering for Real Sampling

The filter designed from the real-sampling image-folding requirement used 32 taps. The passband variation was within 0.01 dB, and attenuation at the folding frequency was 145 dB.

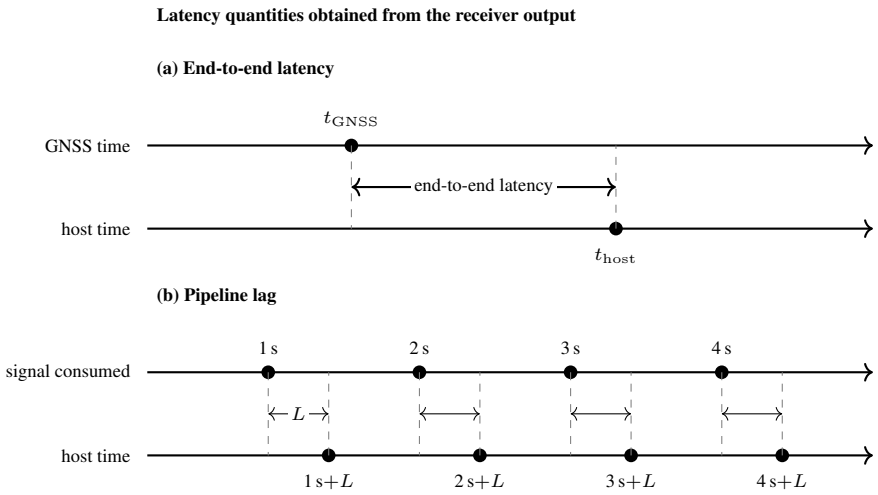

Figure 3. Latency measurements used in the transport comparison. End-to-end latency is measured from the GNSS-derived measurement epoch to the host time at which the corresponding position line appears. Pipeline lag compares the receiver time consumed with the host time at which the corresponding progress line appears.

For comparison, a five-tap filter sufficient for a zero-IF I/Q configuration provided only 17 dB attenuation at the relevant frequency. Such a filter would not adequately reject the image introduced by the real-sampled configuration.

The result demonstrates that the PocketSDR sampling mode must be considered when configuring the downstream signal-processing chain. A filter designed only from the desired passband can be insufficient when a real-sampled configuration produces an image that folds during subsequent decimation.

4.4. Direct versus ZeroMQ Transport

The direct and ZeroMQ sources were compared using GPS L1 C/A at 8 Msps. The downstream GNSS-SDR configuration was byte-identical between the two runs. Each run lasted approximately 300 s, and the measurements were collected consecutively under the same sky conditions.

The direct source consumed 178.9 % of one CPU core on average, compared with a combined 205.1 % for the ZeroMQ configuration. This corresponds to a 14.6 % increase in total CPU use for the external transport. GNSS-SDR itself used 12.9 % more CPU in the ZeroMQ configuration, while the separate helper process used an additional 13.3 %.

Both configurations kept pace with the incoming signal. The measured lag drift was 0.00 ms/s for the direct source and -0.01 ms/s for the ZeroMQ source. The lag spread, however, was 58 ms for the direct source compared with 236 ms for ZeroMQ.

The direct source also produced a lower median end-to-end latency, 320.6 ms compared with 334.8 ms. The difference was 14.2 ms. More significant variation appeared in the tail of the latency distribution. The maximum latency was 374.9 ms for the direct source and 679.7 ms for ZeroMQ.

Table 2. Comparison of the direct PocketSDR signal source and the ZeroMQ signal source. GPS L1 C/A at 8 Msps, byte-identical downstream configuration, two 300 s runs under the same sky. CPU is normalized so 100 % represents one CPU core.

	`Pocket_SDR_Signal_Source`	`ZMQ_Signal_Source`
GNSS-SDR CPU	178.9 %	191.8 %
Helper process CPU	—	13.3 %
Total CPU	178.9 %	205.1 %
Lag drift	0.00 ms/s	-0.01 ms/s
Lag spread	58 ms	236 ms
Wire rate, 1 channel	16 MB/s	64 MB/s
Ring-buffer overruns	0	0
Position fixes	13 431	13 167
Time to first fix	25 s	29 s
Latency, median	320.6 ms	334.8 ms
Latency, p95	339.5 ms	353.0 ms
Latency, minimum	287.4 ms	302.1 ms
Latency, maximum	374.9 ms	679.7 ms

Approximately 300 ms of the observed latency was common to both configurations and therefore corresponds to receiver processing rather than the transport mechanism. The process boundary therefore produced a relatively small change in typical latency but a larger change in worst-case latency and lag variation.

The transport also changes the amount of data moved be-

tween processes. The direct source retains the compact PocketSDR representation until the samples enter the GNU Radio processing graph. The ZeroMQ implementation converts the samples to `gr_complex` before transport. At 8 Msps, this increases the one-channel transport rate from 16 MB/s to 64 MB/s. At 20 Msps, the corresponding rate would be 160 MB/s.

These results show that the principal difference between the two architectures is not whether either can keep up with the signal. Both did so in the evaluated runs. The direct source instead reduces the additional processing and transport required between the RF front-end and GNSS-SDR, while providing explicit visibility into front-end ring-buffer overruns.

4.5. Two RF Channels at Once

The multi-channel adapter was evaluated using two simultaneously configured RF channels. The receiver reported:

```
out0 <- CH1: LO=1573.420006 MHz I 2 bits
out1 <- CH2: LO=1225.599998 MHz I 2 bits
RF Channels: 2
```

The two channels were demultiplexed from the same USB stream and connected to separate signal conditioners. During a 600 s run, GNSS-SDR produced 1120 position fixes with zero ring-buffer overruns while both channels were streaming and both signal conditioners were operating.

The experiment demonstrates that the adapter can expose two independently tuned RF channels from one PocketSDR device within the same GNSS-SDR process. Because the channels originate from the same USB stream, their sample ordering remains aligned at the source output.

A second two-channel configuration exposed an important operational limitation. Tracking channels assigned to the second RF channel performed a blocking acquisition over a wide search range. The resulting scheduler stall prevented the flow graph from servicing the incoming front-end stream quickly enough and caused the PocketSDR ring buffer to overrun.

That run produced 2695 overruns during 296 s and no position fix. The satellites in view were measured at 38 – 44 dB-Hz, while total processor utilization remained approximately one third of available capacity. The failure was therefore not explained by weak signals or insufficient aggregate CPU capacity. The overrun counter directly identified the data-path stall.

After acquisition was changed to a non-blocking operation and the number of simultaneously searching channels was reduced, the overruns disappeared. This experiment demonstrates the practical value of exposing the front-end overrun counter: a receiver failure that would otherwise appear as loss of tracking can be associated directly with a transport-level sample-loss event.

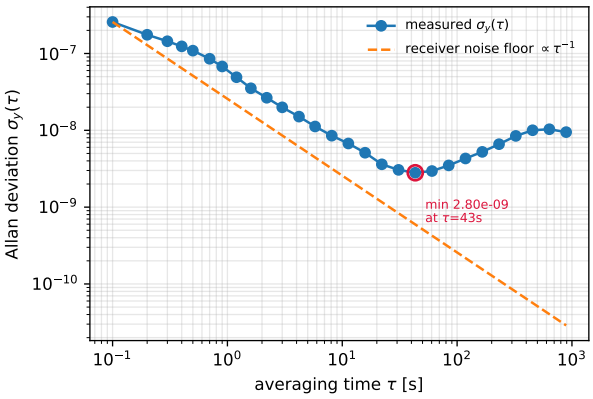

Figure 4. Allan deviation of the PocketSDR TCXO derived from the receiver clock offset. The measurement uses an uninterrupted 37.6 min solution with $\tau_0 = 0.1\,s$. The dashed curve represents the estimated receiver white-phase-noise floor.

4.6. Reference Oscillator Characterization

The integrated receiver was also used to characterize the PocketSDR reference oscillator. Figure 4 was generated from a 37.6 min solution containing 22 584 epochs at 0.1 s. Approximately 1.39 % of the epochs were interpolated across dropped epochs.

The mean fractional frequency offset was -2.65×10^{-7}, corresponding to -0.265 ppm. An independent earlier run produced -0.274 ppm, providing agreement within approximately 4 %.

The Allan deviation decreased from 4.7×10^{-7} at $\tau = 0.1\,s$ to a minimum of 2.80×10^{-9} at $\tau = 43.2\,s$. It then increased again and reached approximately 1×10^{-8} at $\tau = 600\,s$.

The minimum therefore occurs at an averaging time of approximately 40 s, with a measured stability floor of approximately 3×10^{-9}. At short averaging times the receiver measurement noise is more significant, while the longer averaging-time behavior is dominated by the reference oscillator and its drift.

The receiver noise floor is below the measured Allan deviation throughout the evaluated range. At $\tau = 43\,s$ the measured value is approximately five times the estimated receiver floor, and the separation increases at longer averaging times. This supports interpreting the longer-term result as a measurement of the front-end oscillator rather than solely the GNSS receiver measurement noise.

The single-frequency solution nevertheless introduces an important limitation. Ionospheric delay is partially absorbed into the estimated receiver clock offset. This effect becomes increasingly relevant at long averaging times, where oscillator drift is also being evaluated. A future implementation using an ionosphere-free combination of two GNSS frequencies could reduce this contribution. The multi-channel operation demonstrated in the previous section provides the RF-channel capability required for such an extension.

5. Discussion

The experiments establish three distinct properties of the integration.

First, `gr-pocketsdr` provides a direct live path from the PocketSDR USB stream into GNU Radio and GNSS-SDR. The 20 Msps streaming experiment demonstrates that the source can sustain the front-end data rate while the receiver simultaneously produces navigation solutions.

Second, the direct integration changes the behavior of the data path compared with an external ZeroMQ transport. Both architectures kept up with the signal in the controlled comparison, so the result should not be interpreted as a requirement for ZeroMQ to be replaced in every application. The measured differences are instead associated with the additional serialization, process boundary, and queue introduced by the external transport. These effects were visible in CPU use, lag variation, data rate, and worst-case latency.

Third, the multi-channel implementation provides a single receiver process with access to multiple RF channels originating from the same USB stream. This is important for experiments in which the relative sample timing of multiple GNSS frequencies matters. The overrun experiment also shows that direct access to transport-level status can simplify debugging of real-time receiver failures.

The implementation has limitations. The streaming and transport measurements were obtained on a particular host configuration with the CPU governor set to `powersave`. The latency results therefore characterize the evaluated software and hardware configuration rather than establishing a universal transport latency. Similarly, the oscillator characterization is limited by the duration and continuity of the available GNSS solution.

The real-sampling experiment also illustrates that integrating an RF front-end is not limited to writing a source block. The sampling architecture affects the downstream filtering requirements. In the PocketSDR configuration used here, the image associated with real sampling must be considered before decimation.

Finally, the oscillator experiment demonstrates one of the applications enabled by direct integration. Once the front-end is available as a live source, GNSS-SDR can expose receiver-level observables that can be used to investigate properties of the front-end itself. The same architecture can therefore support both receiver operation and RF-front-end characterization.

6. Conclusions

This paper presented `gr-pocketsdr`, a GNU Radio interface for live streaming from PocketSDR GNSS front-ends, together with its integration into GNSS-SDR. The source receives the PocketSDR USB stream, reconstructs the quantized samples, exposes selected RF channels as GNU Radio streams, and applies the front-end configuration at startup.

The implementation sustained 20 Msps streaming during a 328.6 s GNSS-SDR run with zero ring-buffer overruns and 287 position fixes. The multi-channel extension simultaneously exposed two independently tuned RF channels from one device while preserving the sample ordering established by their common USB stream.

A controlled comparison with a ZeroMQ-based transport showed that both architectures could keep pace with the incoming signal under the evaluated conditions. The direct source nevertheless used 14.6 % less total CPU, reduced median end-to-end latency by 14.2 ms, reduced lag spread from 236 ms to 58 ms, and reduced maximum measured latency from 679.7 ms to 374.9 ms. The direct source also retained visibility into ring-buffer overruns and avoided the fourfold increase in one-channel transport data caused by converting the samples to `gr_complex` before crossing the process boundary.

The two-channel experiment further demonstrated that transport-level observability can help diagnose real-time failures. A blocking acquisition operation produced 2695 ring-buffer overruns and no position solution despite adequate signal strength and available CPU capacity. Changing the acquisition behavior removed the overruns.

Finally, the integrated receiver was used to characterize the PocketSDR reference oscillator. The measured Allan deviation reached a minimum of approximately 3×10^{-9} at an averaging time of approximately 40 s, with the longer-term behavior separated from the estimated receiver noise floor.

The software is available at https://github.com/minhaj6/gr-pocketsdr, and the `Pocket_SDR_Signal_Source` adapter is available in the GNSS-SDR upstream development branch.

References

Fernandez-Prades, Carles, Arribas, Javier, Closas, Pau, Aviles, Carlos, and Esteve, Luis. Gnss-sdr: An open source tool for researchers and developers. In *Proceedings of the 24th International Technical Meeting of The Satellite Division of the Institute of Navigation (ION GNSS 2011)*, pp. 780–794, 2011.

Friedt, Jean-Michel. Broadband data transfer over USB for GNU/Linux: 1–2 GHz (L-band) SDR receiver dedicated to GNSS (and other) reception, interfacing with PocketSDR, GNU Radio and gnss-sdr. URL http://jmfriedt.free.fr/fosdem2025.pdf.

Takasu, Tomoji. Pocket SDR: Design, implementation and applications. URL https://gpspp.sakura.ne.jp/paper2005/pocketsdr_seminar_202411_revA.pdf.